Semantic Heap Snapshots - Project Report [October 2020]

Status quo of testsuite

The fork currently fails 13 tests of the testsuite.

testcase	
all/assume/assume10QPwand.vpr	incompleteness, reason unknown
all/issues/silicon/0120a.vpr	timeout, See below *Termination*
all/issues/silicon/0228a.vpr	heap dependent trigger incompleteness (*Amplified Problems*)
all/issues/silicon/0493c.vpr	incompleteness, reason unknown
all/third_party/stefan_recent/testListAppend.vpr	incompleteness, reason unknown
all/third_party/stefan_recent/testTreeWand.vpr	timeout
all/third_party/stefan_recent/testTreeWandE1.vpr	timeout
all/third_party/stefan_recent/testTreeWandE2.vpr	timeout
quantifiedpermissions/issues/issue_0184.vpr	Fixed incompleteness (UnexpectedOutput)
quantifiedpermissions/misc/heap_dependent_triggers.vpr	heap dependent trigger incompleteness (*Amplified Problems*)
wands/examples_new_syntax/ListIterator.vpr	unstable incompleteness, sometimes passes, reason unknown
wands/new_syntax/QPWands.vpr	incompleteness, reason unknown
wands/new_syntax/SnapshotsNestedMagicWands.vpr	incompleteness (*Amplified problems*)

Fixed problems

- **silicon#486**: The upstream function translation can result in invalid trigger expressions (`ite`), as well as using undeclared variables in the function body. No similar problems have been found in the fork so far and the example above verifies without SMT warnings.
- **silicon#184**: reason not investigated

Amplified problems

- **SMT definitions inside package scope** do not survive and result in large incompletenesses. This problem already exists in upstream and was known before. My fork amplifies this since `produce(a1 && a2)` results in two new snapshot symbols whose definition is required to use them. As an example, the `SnapshotsNestedMagicWands.vpr` fails because the wand LHS contains a conjunction that produces fresh symbols whose definitions are unavailable outside of the package scope.
- **Heap dependent triggers** for Silver-quantifiers have had unstable behavior in the past (silicon#204, silicon#371). The fork produces a different triggering behavior from upstream in some cases. The reason for this are most likely some snapshots that were syntactically equal before but no longer are and thus prevent a trigger from matching.

Silver Language Support

The fork supports inhale-exhale expressions in predicate bodies and function preconditions. Upstream rejects such programs, two of which are added to `all/inhale_exhale` tests and illustrated below:

```
field f : Int
field g: Int
```

```
predicate pair(x: Ref)
{ [acc(x.f) && acc(x.g), acc(x.g) && acc(x.f)] }

function fun(x: Ref): Int
    requires [acc(x.f) && acc(x.g), acc(x.g) && acc(x.f)]
{ x.f }

method swap_predicate_body(x: Ref)
{
    inhale acc(x.f) && acc(x.g)
    x.f := 0
    x.g := 1

    fold pair(x)
    unfold pair(x)

    assert x.f == 0
}


method swap_function_precondition(x: Ref)
{
  inhale acc(x.f) && acc(x.g)
  x.f := 0
  x.g := 1
    assert fun(x) == 0
}
```

Maintainability

Function Recorder

The `FunctionRecorder` is removed from the fork such that function verification and function axiom are completely decoupled. For illustration, here are some points about the commit [1] which removes the function recorder:

- Removed references to the function recorder on 190 lines 13 files
- Deleted the 250 lines of the function recorder itself
- Evaluator profited most, many cases got considerably simpler
- Manual "threading-through" code no longer pollutes unrelated modules like StateConsolidator

More expressive function axioms

SMT function axioms closer to Viper und thus hopefully easier to debug. Snapshot (heap) locations within the function body are referenced explicitly as opposed through a complicated indirection through the syntactic structure of the precondition. Additionally, `unfolding`'s are now also reflected in snapshots structure and function axioms.

Snapshot unification

Upstream uses three snapshot concepts (and SMT sorts + axiomatizations):

1. `Snap`: simple, cheap, structural / syntactic
2. `FVF`: expressive, expensive, semantic
3. Quantified predicate snapshots

The fork extends `FVF` to `PHeap` which additionally supports predicates. Additionally, it contains pre-defined versions simple snapshots, i.e. singletons corresponding to an `acc(resource)` plus rewriting axioms that simplify those cases directly. This makes case 1 above still reasonably simple (in terms of how many axioms need to be instantiated etc), although these also use the much richer snapshot structure `PHeap` instead of the very efficient `Snap`.

Soundness

From the test suites perspective, no observable unsoundness was introduced

Completeness

- All of the 13 test failures described above are either incompletenesses or timeouts, so currently the fork is less complete than upstream.
- An incompleteness in `issue_0184.vpr` was fixed
- A small snapshot incompleteness with conditional permissions and predicates is removed. This fails in upstream and verifies in the fork (simplified from a case added to test `quantifiedpermissions/issues/issue_0205.vpr`):

```
predicate F(x: Ref)

function fun01(y: Ref, b: Bool): Int
  requires acc(F(y), b ? write : none)

method test01(y: Ref)
{
  assert fun01(y, false) == old(fun01(y, false))
}
```

Termination

Matching loop

A function with a wildcard-unfolding results in an SMT theory with a matching loop (4.4.1 in my thesis [3])

Examples

- `silicon/0120a.vpr`: In this case the solver reliably exploits the loop and never terminates. A **Viper level workaround** that works here is to simply replace the wildcards in functions with write permissions, but obviously that is not a general solution.
- `functions/wildcards.vpr`: I have added this testcase to illustrate the scenario in which a function makes use of the wildcard-unfolded predicate and thus relies on a more precise semantics being implemented.

Implemented Approximation The approximation/heuristic used for now is very coarse-grained. For this function (`p` is a metavariable):

```
function f() {
  unfolding acc(P(args), p) in e
}
```

The body translation algorithm will:

1. If `p` syntactially is `wildcard`, then it keeps `P(args)` in the snapshot used to evaluate `e`
2. Else it removes the instance from the snapshot used to evaluate `e`

That means in case 1 the snapshot is exact and in 2 it is a lower bound.

Increased SMT preamble length

For programs with many Viper resources the SMT preamble emitted by my fork is much longer than upstream's. This comes from my snapshot axioms for polymorphic heaps that need to describe every snapshot constructor's properties with respect to every resource. For example:

```
dom_PREDICATE(singleton_FIELD(...)) = {}
...
dom_OTHERFIELD(singleton_FIELD(...)) = {}
...
```

In severe cases this leads to preambles of 100k lines wheres upstream only produces 1k. Part of this preamble is the axiom for extensional equality of two snapshots, which becomes one single large term for programs with many resources, see [4] for an example with a 1000+ lines axiom.

Internal timeout with z3-4.8.9

Upstream does not appear to have this problem and no time was invested in finding a workaround.

- https://github.com/Z3Prover/z3/issues/4713

Performance

Two benchmarks were done using by extending the `PortableSiliconTests` class. Each program is verified 5 times and if it yields stable results, the middle 3 runs are taken to compute the mean verification time. Programs are taken from the frontend and Silicon's own testsuites so they can contain expected failing assertions. It is not clear how to compare runs with different verification outputs, so only ones with equal outputs were compared.

Frontend benchmark

This is a set of 72 Viper programs generated by Prusti, Vyper, Gobra and Nagini chosen by manual "random" selection looking for large files. All programs can be found in `benchmark-20-10-07/frontend-generated/`.

- stable upstream: 71
- stable fork: 66
- equal verification results: 64

Testsuite benchmark

The second benchmark is Silicon's testsuite with a few additional tests developed during the project, a total of 1060 programs.

- stable upstream: 1032
- stable fork: 1030
- equal verification results: 982

Results

- The two Silicon versions produce stable equal verification outputs for 1046 Of the 1132 tested programs.
- Average slowdown of 30%
- Average slowdown of 34% when ignoring programs that verify in less than 1 second in upstream
- These averages are dominated by the testcases which verify in 1-5 seconds which form the bulk of all considered programs
- It looks like that there is a correlation between increased preamble length and slowdown, but to me it does not look like the slowdown is primarily caused by this (See figure 3). Without correcting for absolute verification time, there is no correlation between slowdown and preamble length.

For more details, consider figures 1 and 2.

Limitations & future work

Remove FVF embedding

- **time estimation:** 2 days - 1 week

QP Embedding can be removed since FVF is a subset of PHeap. First a lot of simple work replacing and removing old code, then probably some difficult debugging because the behavior will most likely not be 100% identical (syntactic changes of terms might lead to triggering issues).

Permissions in functions

- **time estimation:** flexible, depending on how precise the permissions are represented. I refined my initial approximation with a few lines of code in a few hours, but encoding precise permissions into snapshots would probably take several weeks.

Permission representation in functions is a coarse-grained approximation which can be exploited for incompletenesses with artifical examples (none from the testsuite do). I tried but failed to produce an unsoundness based on this approximation, but cannot exclude it for sure. Note that upstream is also not perfect at this and it seems that both versions have slightly different limitations as to how precisely permissions in functions are represented. (For example see *Completeness*).

Improve extensional equality

- **time estimation:** cannot predict, but my unsuccessful experiments took ~2 days

Try to reduce extensional equality instantiations for improved performance. Main challenge here is to retain functionality of the predicate-based function triggers. I attempted some ad-hoc solutions, but they are not functioning correctly and result in many timeouts (`move-restrict-f` and `move-restrict-f-subheap`)

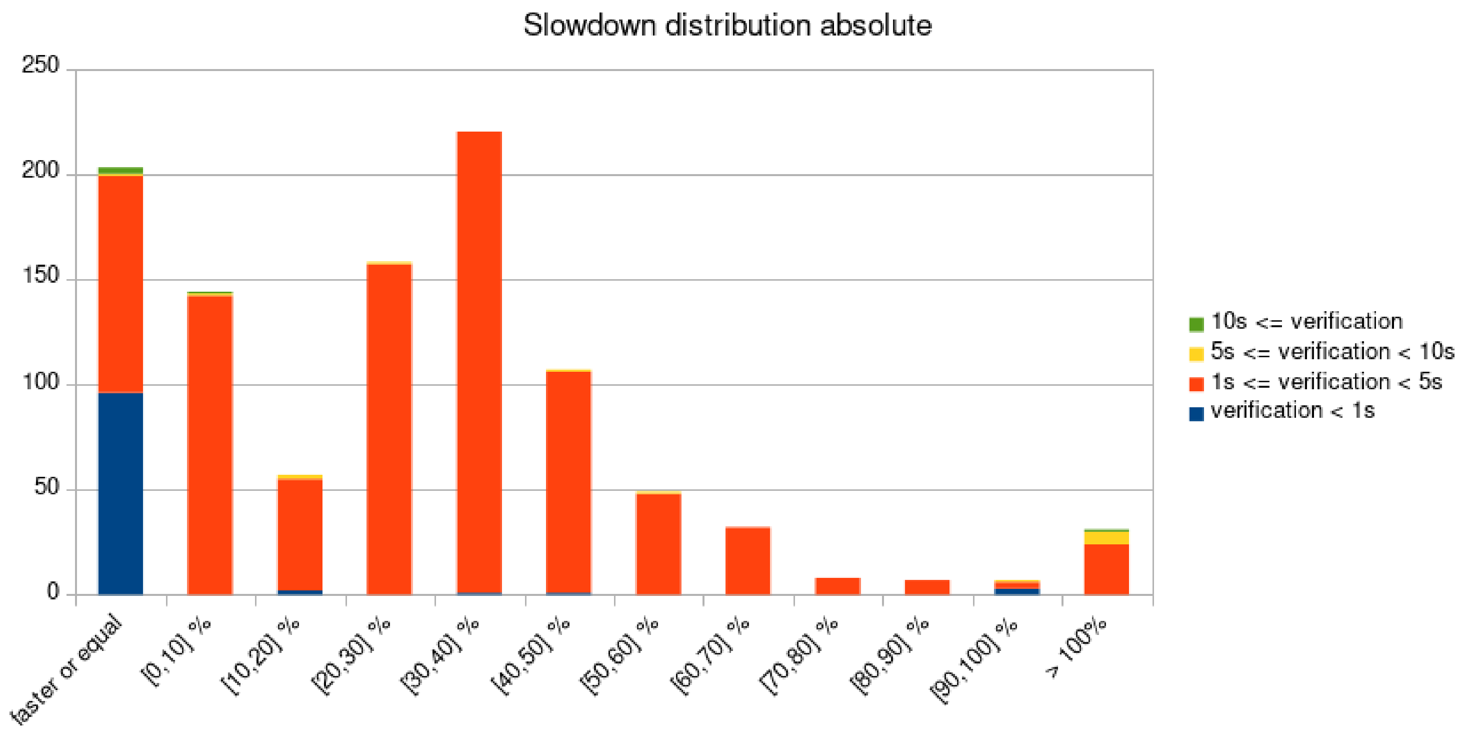

Figure 1: Number of programs that are slowed down by a certain percentage, grouped by their absolute verification time in upstream.

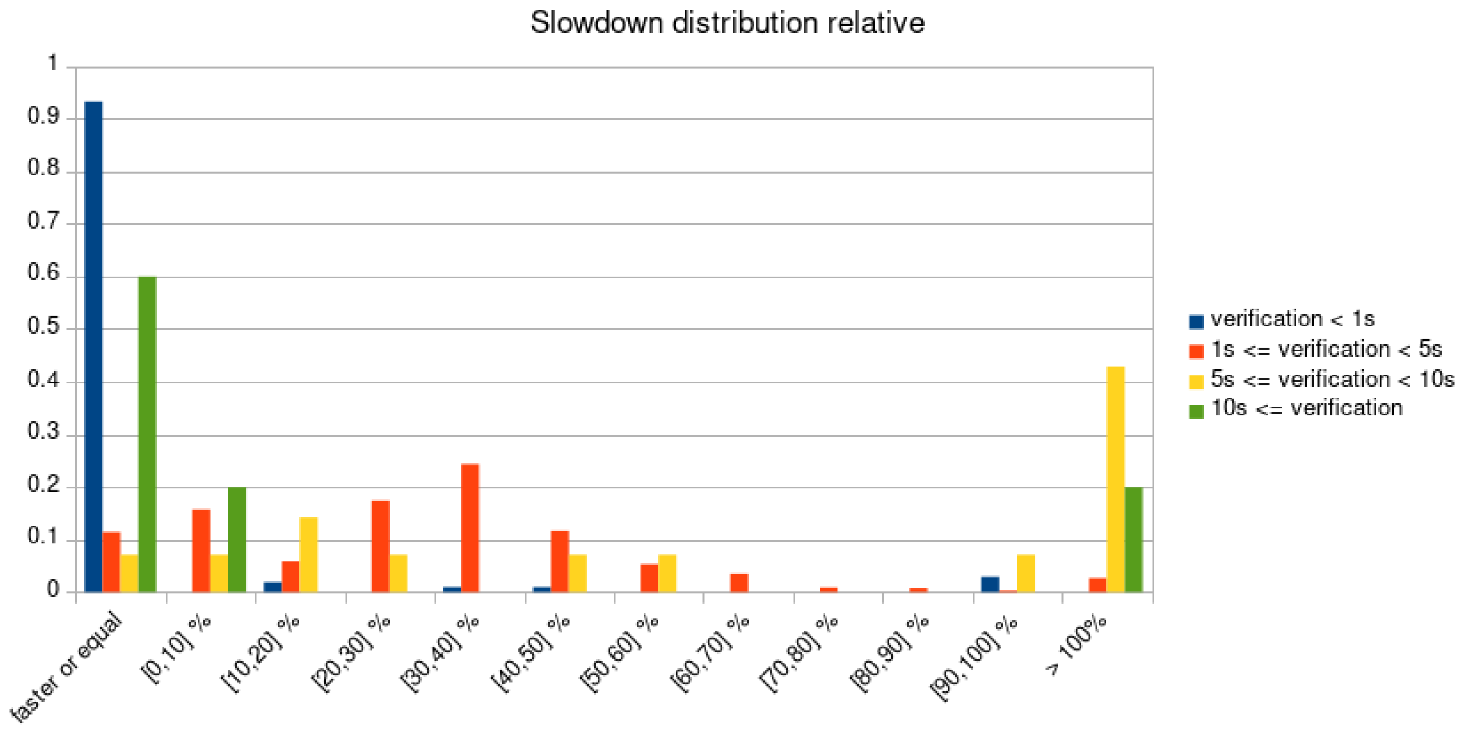

Figure 2: Slowdown distribution normalized to the sizes of the four groups, i.e. the percentage of each group that falls into a certain interval of slowdowns.

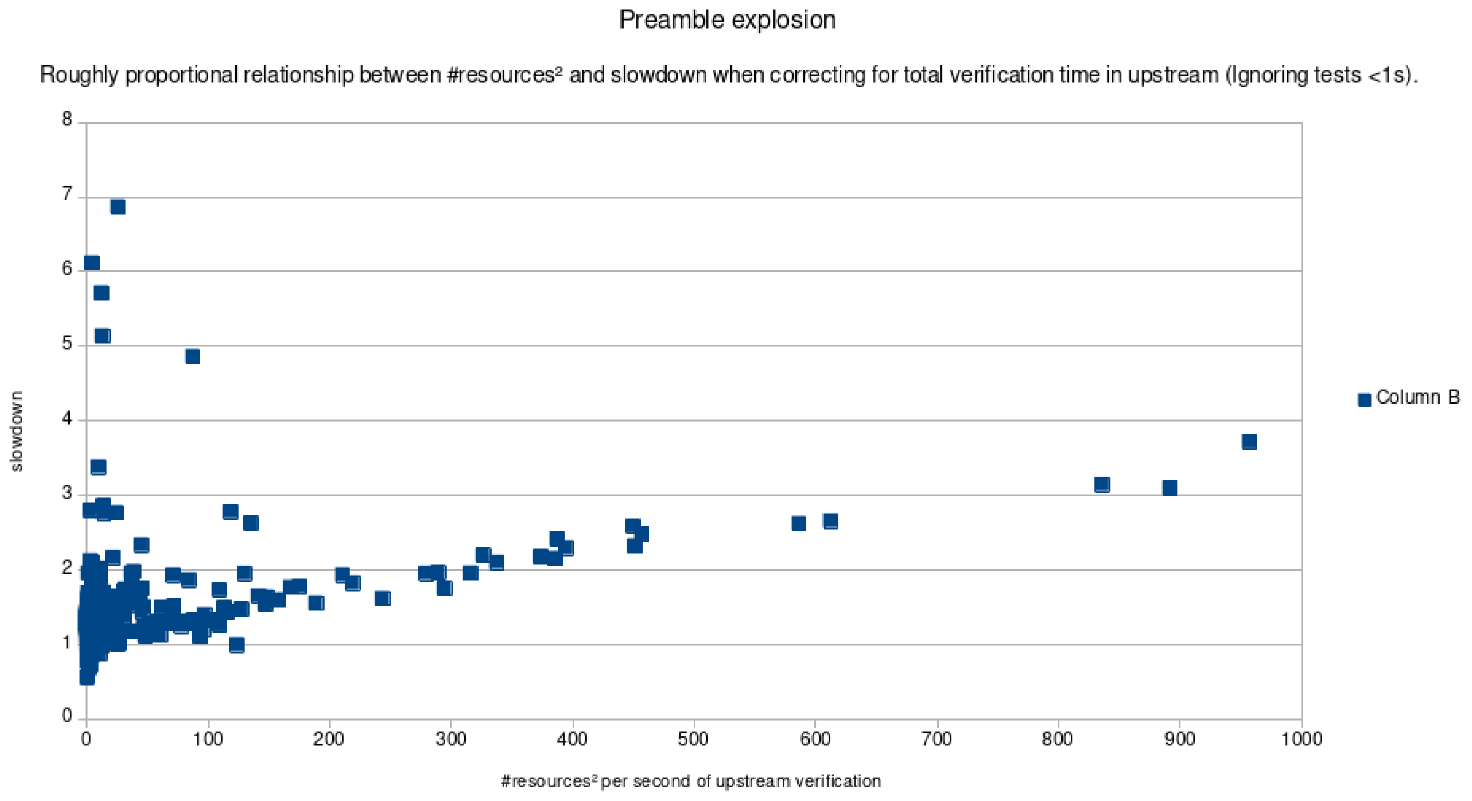

Figure 3: Correlation between SMT preamble length and slowdown.

Additionally, the axiom for extensional equality itself can get very big (some prusti testcases produce up to 200 resources). The axiom contains a nested quantifier for each resource ('should be equal on all lookups of this resource') and maybe alternative encodings of this axiom more efficient.

Clean up ad-hoc tuple axiomatization

- **time estimation:** 1 week

One of the introduced sorts `Loc` is used to represent n-tuples for predicate arguments (size statically known: arity of predicates). This could be replaced with a more generic n-tuple sort potentially using built-in Z3 datatypes. That means implementing a supporter which produces the axiomatization for the tuple-sorts, based on the predicates in the program under verification. Then both static axioms (SMT files) and dynamically generated ones as well as the QPP code need to be adapted.

Flat produce

- **time estimation:** Difficult to predict since I do not know how I would do this, but implementation of a correct solution would probably take less than a day.

Flat `produce`: Currently, producing a conjunction produces two new snapshot symbols. This was not required before, because the snapshot could be destructured:

```
// upstream SE-rule
produce(a1 && a2, h):
  produce(a1, First(h))
  produce(a2, Second(h))
```

However, this is no longer possible because semantic heap snapshots no longer guarantee this structural mapping between assertion and snapshot. Thus, the rule produces fresh symbols (skolemized existentials):

```
// fork SE-rule
produce(a1 && a2, h):
  h1,h2 := fresh(PHeap)
  produce(a1, h1)
  produce(a2, h2)
  assume(h = h1 * h2)
```

This makes it more work to read SMT logs and causes trouble in some situations where definitions of symbols are not properly retained (e.g. the `package` scope problem). By changing the inner workings of `produce` one might be able to get rid of these additional symbols.

Monomorphic heaps

- **time estimation:** 2-4 weeks

Some literature suggests that monomorphic heap representations (one heap per field) are more efficient than polymorphic ones [2]. I have not tried this, but it might be worth it. Main challenge is one of software engineering, i.e. to minimize the parts of code that need re-writing.

---

[1] https://github.com/maurobringolf/silicon/commit/25777a43853c827d1458c5882f9a31049a5d8342.

[2] Böhme, Sascha, and Michał Moskal. "Heaps and data structures: A challenge for automated provers." International Conference on Automated Deduction. Springer, Berlin, Heidelberg, 2011.

[3] Mauro Bringolf, BA thesis, Towards better Function Axiomatization in a Symbolic-execution-based Verifier, https://ethz.ch/content/d
interest/infk/chair-program-method/pm/documents/Education/Theses/Mauro_Bringolf_BA_thesis.pdf

[4] https://gist.github.com/maurobringolf/3608335209004945cb7d1778fcb111bb#file-error-smt2-L6845-L8036